Metadata-driven library design

Antonio Cisternino
Dipartimento di Informatica
L.go Bruno Pontecorvo, 3
I-56127 Pisa, Italy
+39 050 2213149
cisterni@di.unipi.it

Walter Cazzola
Dipartimento di Informatica e Comunicazione
cazzola@dico.unimi.it

Diego Colombo
IMT Lucca
colombod@di.unipi.it

ABSTRACT
Library development has greatly benefited form the wide adoption of virtual machines like Java and Microsoft .NET. Reflection services and first class dynamic loading have contributed to this trend. Microsoft introduced the notion of *custom annotation*, which is a way for the programmer to define custom metadata stored along reflection meta-data within the executable file. Recently, Java also has introduced an equivalent notion into the virtual machine. Custom annotations allow the programmer to give hints to libraries about his intention without having to introduce semantic dependencies within the program; on the other hand these annotations are read at runtime introducing a certain amount of overhead. The aim of this paper is to investigate the impact of this new feature on library design, focusing both on expressivity and performance issues.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Contructs and Features – *reflection.*

General Terms
Languages.

Keywords
Reflection, Code Annotation.

1. INTRODUCTION
Reflection and dynamic loading are becoming essential elements of modern programs. Their usefulness is testified, for example, by the JDBC architecture that shows how to implement a driver based architecture exploiting Java's mechanism for dynamic loading.

Although reflection can be used to inspect the structure of types, to access fields, and even to invoke methods dynamically, the concept of tagging has been anticipated as an interesting application. Consider for instance the Java serialization architecture: the programmer can declare the instances of a *serializable* class simply by implementing the `Serializable` interface, which in fact is an empty interface. Thus two types that differ only in the implementation of the Serializable interface are indistinguishable from the execution standpoint. Besides, the serialization of the instances of non-serializable types will not be allowed by the serialization support. Java serialization taught us that the metadata stored with the code can be used for other purposes than mere execution. Other programs may rely on the reflective abilities of inspecting the compiled types and act differently depending on what they have found.

Although widely used by Java programs, the idea of providing explicit metadata support for annotation has been introduced by Microsoft in the Common Language Runtime (CLR). The virtual execution environment is part of the CLI standard [1, 2]. More recently also Java introduced annotations as a mean of storing custom data inside Java classes [3]. There are also proposals to add extensible reflection to C++ language [4].

Custom annotations have shown to be useful because they provide a channel that library-users and library-developers may use to communicate. A library may require that the user puts annotations on top of classes and methods in order to instruct the library on how to use it.

Unfortunately, the availability of this new mechanism increases the number of possible choices a library developer has for modeling the abstractions to be provided to the final user of the library. The choice of using custom annotations instead of more traditional programming abstractions should be subjected to consideration about expressiveness and performance issues.

The paper is organized as follows: section 2 introduces custom annotations; section 3 is devoted to discuss how annotations have been used so far in real applications; performance considerations are presented in section 4; section 5 presents conclusions. As a final remark, throughout the rest of this paper we will also refer to custom annotations as *custom attributes* and we will use the C# notation inside the examples.

2. CUSTOM ATTRIBUTES
A *custom attribute* is a piece of information attached by the programmer to a portion of a program. In the model implemented both in Java and .NET, attributes can be attached only to those elements accessible through the reflection API, such as assemblies, types (delegates, value types, and classes), fields, properties, and methods; however there has been a proposal of extending the annotation model to code blocks [5].

Conference'04, Month 1–2, 2004, City, State, Country.

In .NET custom attributes are represented by instances of classes that inherit from the system class `Attribute`. Java exposes annotations as instances of an interface.

A custom attribute is defined by specifying a set of values and the type of the attribute; all the values used to create it must be computable at compile time. The following is an example of annotations in C#:

```
[MyAnnotation(“par”, Property=”val”)]
public class MyClass {..}
```

The definition of `MyAnnotation` attribute can be the following:

```
class MyAnnotationAttribute : Attribute {
  MyAnnotationAttribute(string par) {...}
  public string Property;
}
```

Parameters required to instantiate custom annotations are stored inside the binary file, along with the rest of reflection meta-data, so that they can be retrieved at run time. This data is *ignored* by the execution environment unless explicitly accessed through the reflection API. For instance, let `m` be an instance of `MethodInfo` class (a reflective descriptor of a method); in C# we can retrieve the custom attributes associated with the method as follows:

```
Attribute[] attr = m.GetCustomAttributes();
```

The crucial idea behind the custom annotation consists of shifting data about the code into the executable and to be available at runtime. Custom annotations are interpreted by programs and are used for program transformation.

A stereotypical example, from Microsoft .NET, of custom attributes usage is the support for implementing web services by means of custom attributes. `WebMethod` attribute is used to label methods that should be exposed as web services. A minimal web service written in C# that computes the sum of two integers is the following:

```
public class HelloWorldWS {
  [WebMethod]
  public int add(int i, int j) {return i+j;}
}
```

Once compiled, the `HelloWorldWS` class does not provide any web services interface. A different program—actually part of the Internet Information Server — is responsible for looking up reflection information within assemblies and generating a SOAP/WSDL interface to the method `add` over HTTP.

The essence of annotations is that information is stored together with the code so that some other meta-program will need only the executable file to access the information. Although this may seem to be a little change with respect to configuration files shipped with the executable program, it makes all the difference. With annotations the programmer can decorate the program, without having to define bindings between types and custom information. Moreover configuration files are separated from the executable, leading to a weaker link between the code and its configuration. In the past we have dearly paid the separation of the meta-data from the data, as it is still witnessed by the COM [6] architecture in Windows, where metadata are stored inside the disliked system registry.

To better appreciate the effectiveness of custom annotations versus the use of external configuration files it is worth to briefly describe the Java Web Service development pack [7], currently based on Java 1.4 (the Java version prior to custom annotations). With this library the programmer should define several XML configuration files to control the module responsible for generating SOAP/WSDL. For instance the interface of the Web service is defined with an XML document similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-
rpc/ri/config">
  <service
      name="MyHelloService"
      targetNamespace="urn:Foo"
      typeNamespace="urn:Foo"
      packageName="HWS">
      <interface name="HWS.HelloWSIF"/>
  </service>
</configuration>
```

Despite its verbosity the only purpose of the file is to annotate the `HWS.HelloWSIF` interface as a web service (i.e. all the methods of the interface should be considered operations of the service).

3. USING ANNOTATIONS

Libraries were originally conceived as collections of common-use routines that programmers can import within their programs. Today, libraries have become tangled set of programming abstractions (usually in the form of classes) modeling some application domain. To use a library it is required to understand its lingo and how the domain values and operations fit together.

Often libraries are used as a way to extend the programming language with new features (this practice originated with C where even the basic I/O was provided in the form of a library); they contribute to define a language within the language, designed for a given application domain.

In this section we discuss possible uses of custom annotations to support the definition of library interfaces.

3.1 General considerations

Custom attributes allow tagging programming elements; they differ from the inheritance relation in two ways:

1. annotations are parametric, inheritance not (unless some form of generics is taken into account, and even then it is possible only if specialization is available);
2. unlike inheritance that imposes a small amount, though not null, of overhead at runtime, annotations are passive unless explicitly read

Another important aspect of annotations is that they are orthogonal to other relations; therefore they are suitable for introducing new relations among types of a programming language. Attributes are user-defined, thus there is not a predefined set of them, and a library may introduce as many of them as required.

In the area of domain specific languages custom attributes are useful to define the traits of types [8]. Traits are used to configure a generic library so that the amount of information is enough to specialize it to some particular application. In the context of generative programming traits are usually processed at compile time,

along with program specialization. At the moment, custom annotations are processed at run time, introducing possible overheads that could be in principle avoided. We will further discuss this issue in the next section.

Custom annotation cannot refer directly to objects that will be available at runtime. This is required because the arguments passed to an annotation's constructor should be evaluated at compile time, in a different context of the runtime.

3.2 Serialization

Serialization is the process of writing a structured object in a serial stream. As we pointed out in the introduction, serialization originated the idea of using interfaces for tagging classes in Java.

With custom attributes it is possible to go further and control the whole process of serialization of instances of a given class. Let us consider the following example:

```
[XmlRoot("NewGroupName"),
 XmlType("NewTypeName")]
public class Group{
    [XmlArrayItem("MemberName")]
    public Employee[] Employees;
}
```

In this case the class `Group` has been annotated to indicate how its instances should be serialized. The root element will be named as indicated, the same will happen for XML type name that will be used within the associated XSD schema. More interesting is the annotation over the `Employees` field, which indicates that in the serialized array only the `MemberName` fields of Employee instances must be serialized. Thus in the serialized structure we will only partially serialize the associated employees.

3.3 Indigo and Web Services

We already discussed in the previous section how attributes can be used for defining Web services. A class defines a Web service, and annotated methods indicate the methods that should be exposed as operations.

The upcoming library codenamed Indigo [9] for supporting distributed computations based on web services standards heavily relies upon custom annotations. The library revolves around the notion of *data contract* and *service contract*. As we might guess from the names, the first refers to the structure of the data as it is seen from outside of the application, the second to the definition of published operations.

Here is a simple example of data contract:

```
[DataContract]
public class Person
{
    [DataMember]
    public string fullName;
    [DataMember]
    private int age;
    private string mailingAddress;
    private string telephoneNumberValue;
    [DataMember]
    public string TelephoneNumber
    {
        get {return telephoneNumberValue; }
        set {telephoneNumberValue = value; }
    }
}
```

The traditional approach to marshalling in frameworks like CORBA [10], Java RMI [12], and .NET remoting [13], is to define a type so that its serialized form coincides with the message to be sent on the network in inter-process communications; in this way we let the run time take care for us of the communication.

Using custom attributes, Indigo decouples the data structure from its serialized form required for network communications. This is possible because, as we already said, custom attributes define an orthogonal dimension to that of the type system.

In the example above only the members labeled `DataMember` will be serialized in communications (even if they are private inside the process!). The same approach is used for defining data contracts:

```
[ServiceContract]
public interface IOne
{
    [OperationContract(IsOneWay=true)]
    void A();
}
```

Service contracts provide information about how methods should be exposed to network users of the service. Annotations allow us to provide additional information on the behavior of the particular operation, in this case the fact that the operation will not return any value so that the client can close the connection as soon as possible.

A similar approach has been taken by Robotics4.NET [11], a software library supporting the development of control software for robotics systems. In this case annotations are used to define incoming and outgoing messages from a sort of agent, called roblet. Custom annotations are used by the framework to implement the communication infrastructure among the roblets and the control software of the system. The following is an example of such a roblet:

```
namespace HeartBeat {
  public class Beat : RobletMessage {
    public long tick = DateTime.Now.Ticks;
  }

  [OutputMessage(typeof(Beat))]
  public class HeartBeatRoblet : Roblet {
    public HeartBeatRoblet() : base("HB") {}
    protected override void Run() {
        SendState(new Beat());
    }}}
```

The `SendState` method is responsible for taking care of message dispatching, and its behavior is controlled by the custom annotations indicating friendship among agents, input and output message types.

3.4 Relational Interface to Databases

In [4] it is discussed how to extend C++ with reflection support by means of template meta-programming techniques. The proposed reflection system provides support for custom meta-data.

The paper discusses how a library for building search engines can benefit from the declarative power of custom attributes. In this case attributes drive storage information of the objects:

```
class DocInfo {
  char const* name;
  char const* title;
```

```
  int        date;

  META(DocInfo,
       (FIELD(name,
              (MaxLength(256),
               IndexType(Index::primary))),
        FIELD(title, MaxLength(2048)),
        FIELD(date, IndexType(Index::key)));
};
```

Similar to C#, attributes are objects stored within the meta-class. In this example we use `MaxLength` and `IndexType` attributes to control how the search engine library must store and index objects on the secondary storage.

3.5 Code Annotations

Assuming custom annotations capable of annotating portions of code as it is done in [a]C#, an extension to the C# language, we can use them for more finer grained tasks.

Using this kind of annotation it is possible to annotate a code with hints on about how to produce the concurrent version of it:

```
public void m() {
 [Parallel("Begin of a parallel block")] {
    Console.WriteLine("Main thread code");
    [Process("First process")]
    { /* Computation here */ }
    [Process]
    { /* Computation here */ }
  }
  Console.WriteLine("Here is sequential");
}
```

In this case we rely on annotations to mark `Parallel` a block of code. Inside we define code blocks annotates as `Process` that can run in parallel.

3.6 Attribute Usage

Microsoft .NET defines a set of “meta-attributes” that can be used as annotation when defining an attribute class. These annotations are used to possibly constrain the attribute usage. The following example defines an attribute that can be used only once and only on classes:

```
    [AttributeUsage(AttributeTargets.Class,
                    AllowMultiple=false)]
    class ClassTgtAttribute : Attribute {}
```

In a sense, the ability of specifying that an attribute can be used only on classes or methods, if it is inherited or not, provides a means for specifying a sort of a customizable syntax for custom attributes.

3.7 Designer Environments

Microsoft Visual Studio [14] designer is capable of loading arbitrary components during the design process of user interfaces. At design time components are configured by specifying a subset of properties that the component should have at runtime.

Microsoft .NET controls can indicate to the designer which properties can be configured at design time by means of custom attributes. Default values of design-time properties are also specified through custom attributes.

The designer is able to display a preview of the component while designing an interface. A custom attribute specifies which class is responsible for generating the preview of a component. The designer, however, should inherit from a specific class in order to be eligible for its role.

Java designer also relies on reflection information in order to load components into the designer. However, in this case a naming convention is used to determine properties to be shown inside the designer. The naming conventions used by Java are defined by the Java Beans specifications.

3.8 Final Considerations

In this section we presented several applications of custom attributes. We believe that many others are possible, making extensible metadata an important tool in the library-designer toolbox.

In particular we believe that the declarative aspect of the approach allow library developers defining interfaces both operational and declarative.

Custom attributes have almost no drawbacks: they allow defining arbitrary relations among data types, are distributed with executables, and always accessible through the reflection API. However there is a noticeable exception: there is the risk of a possible overhead, due to the fact that meta-data interpretation is often performed at run time. In the next section we will discuss this aspect.

4. ABOUT PERFORMANCE

Performance is always important, and custom attributes should not impose a significant overhead over a computation in order to be really useful.

At a first glance it might be evident that meta-data can be retrieved only to runtime through reflection. This implies that, if attributes are used to specify traits of a library, we must postpone computations that could be done at compile time, to runtime.

This is true for the examples shown in the previous section. However it is not true in general: meta-programs can be run before the so-call “runtime”, though they run after the compiler. It is the case of several tools that manipulate binaries available for the various virtual machines.

Nevertheless, when we are interested in using custom attributes directly at runtime, we must consider that the time spent for reading meta-data is not zero. It is however possible to drown this overhead into the overall computations costs: for instance, the Microsoft XML serializer generates dynamically a class for each type it serializes, and annotations are read during this generation process. After this generation phase serialization takes place without any more accesses to custom meta-data.

5. CONCLUSIONS

In this paper we have discussed how custom annotations may affect the design of libraries. The main impact of the mechanism is at the level of library interface; however it also influences the internal design of the library.

Custom annotations provide a means for library users to declare their intentions, and for library developers to better adapt to different uses of the library. There are already libraries taking benefits from their use.

If used in their simplest form annotations require to be processed at runtime. The overhead imposed for accessing them is in general not significant, though it is possible to get rid of it by executing a

meta-program responsible for processing annotations before that the program is executed.

We believe that custom annotations will play a significant role in the design of libraries in the next years, and they will be added to other programming systems that still lack of this kind of support.

6. REFERENCES

[1] James Miller. *Common Language Infrastructure Annotated Standard*. Addison-Wesley, November 2003.

[2] ECMA 335, *Common Language Infrastructure (CLI)*, http://www.ecma.ch/ecma1/STAND/ecma-335.htm.

[3] Java Web Site, Available at: http:// java.sun.com/

[4] Attardi, G., Cisternino, A., *Self Reflection for Adaptive Programming*, Proceedings of Generative Programming and Component Engineering Conference (GPCE), 50-65, LNCS 2487, October 6-8, 2002

[5] Cazzola, W., Cisternino, A., Colombo, D., *[a]C#: C# with a Customizable Code Annotation Mechanism*, In Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05), pages 1274-1278, Santa Fe, New Mexico, USA, on 13th-17th of March 2005. ACM Press.

[6] D. Rogerson, *Inside COM*, Microsoft Press, Redmond, Wa, 1997.

[7] Java Web Service Development Pack Web site, available at: http://java.sun.com/webservices/reference/index.html

[8] C. Myers, *Traits: a new and useful template technique*, C++ Report, June 1995, available at: http://www.cantrip.org/traits.html

[9] Indigo Documentation, available at: http://winfx.msdn.microsoft.com/library/default.asp?url=/library/en-us/indigo_con/html/503fae4b-014c-44df-a9c7-c76ec4ed4229.asp

[10] Common Object Request Broker Architecture (CORBA), http://www.corba.org/

[11] Cisternino, A., Colombo, D., Ennas, G., Picciaia, D., *Robotics4.NET: Software body for controlling robots*, IEE Proceedings Software.

[12] Grosso W., *Java RMI*, O'Reilly, 2001.

[13] McLean S., Naftel J., Williams K., *Microsoft .NET Remoting*, Microsoft Press, 2002.

[14] Microsoft Visual Studio Web Site, available at: http://msdn.microsoft.com/vstudio/